Quicksort

By C. A. R. Hoare

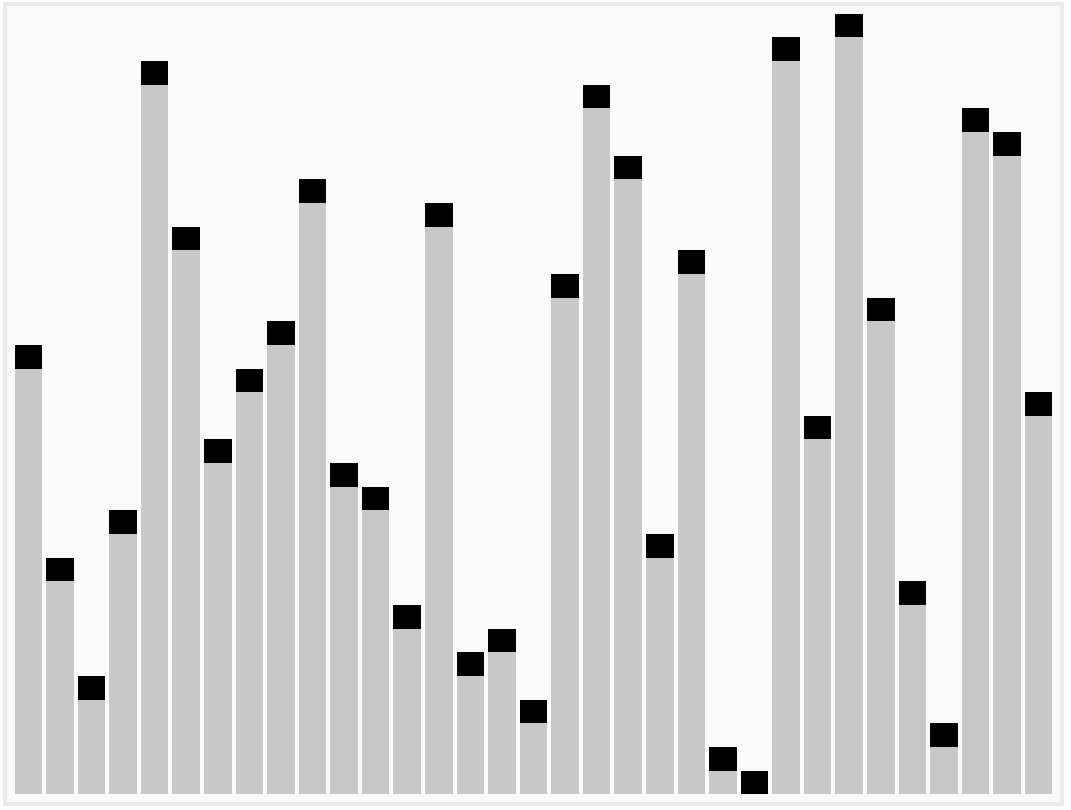
A description is given of a new method of sorting in the random-access store of a computer. The method compares very favourably with other known methods in speed, in economy of storage, and in ease of programming. Certain refinements of the method, which may be useful in the optimization of inner loops, are described in the second part of the paper.

Quicksort

- Quick sort: an efficient, general-purpose sorting algorithm
- Divide and conquer algorithm
- Slightly faster than mergesort for randomized data
  - better cache performance
- Average running time $O(nlogn)$
- Worst-case running time $O(n^2)$
- Despite slow worst-case running time, quicksort is remarkably efficient on average and is often the best practical choice
- In-place sorting algorithm (doesn't require extra space)

Quicksort

- Basic idea:
- Choose an item in the list, and then break the list into two parts
- 1) elements *larger* than item
- 2) elements *smaller* than item
- Recursively sort groups 1 and 2



Quicksort

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

Divide: Partition (rearrange) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1..r]$. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

Quicksort

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5             then i ← i + 1
6                  exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

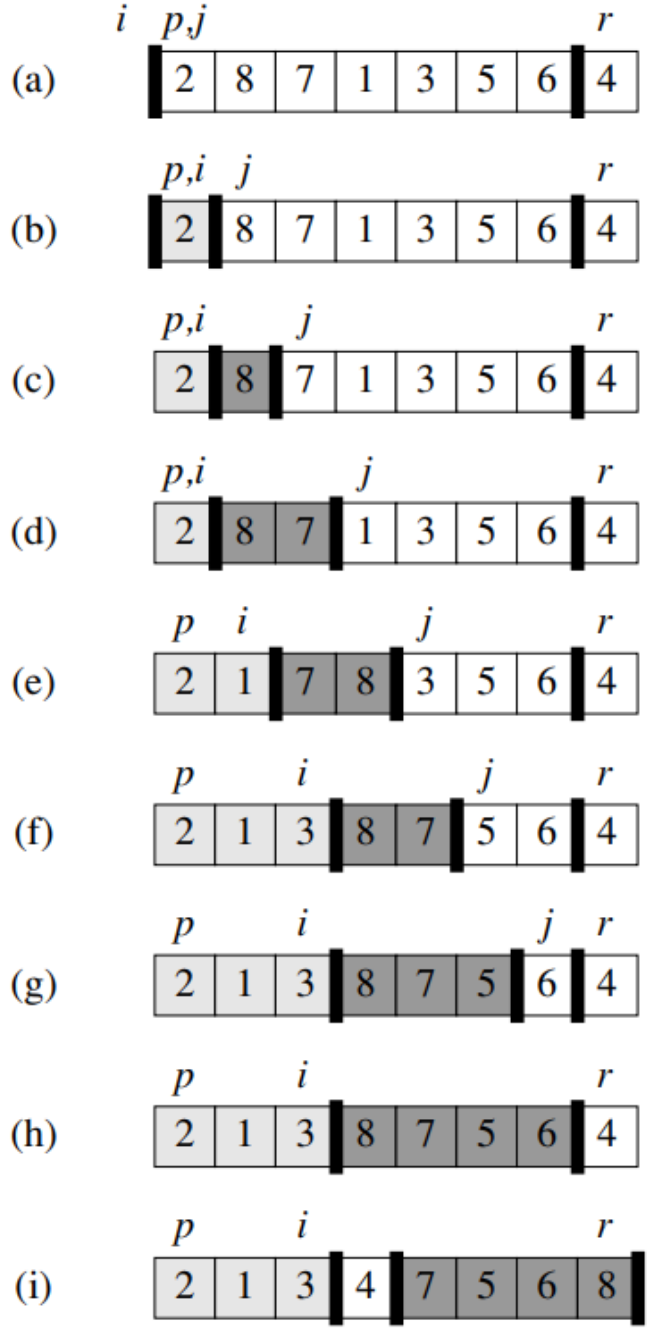


Figure 7.1 The operation of PARTITION on a sample array. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 7 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6 and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

Performance of quicksort

- Performance of quicksort heavily dependent whether the partitioning is balanced or unbalanced, which depends on the input array
- If partitioning is balanced, quicksort runs in $O(nlogn)$
- If partitioning is unbalanced, quicksort runs in $O(n^2)$
- **Worst-case partitioning**: when partitioning routine produces one subproblem with $n-1$ elements and one with 0 elements
- **Best-case partitioning**: partition produces two subproblems each of size $n/2$.
- Average case: much closer to best case than worst case

Performance of quicksort

- Partitioning operation costs $O(n)$, where n is size of subarray being partitioned (size r-p)
- **Worst-case partitioning:**
- $T(n) = T(n-1) + T(0) + \Theta(n)$
  - $\Rightarrow \Theta(n^2)$
- **Best case partitioning:**
- $T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n)$
  - $\Rightarrow O(nlogn)$
- Average-case is much closer to best-case than worse case
- **Example:** assume partitioning always produces a 9:1 split (which seems unbalanced), this gives the recurrence $T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$
- If we draw this as a tree (figure 7.4), it shows the recursion terminates at dept $\log_{10/9} n = \Theta(\log n)$, giving the same overall time complexity as best-case which is $O(nlogn)$
- In fact, any partition of constant proportionality is $O(nlogn)$

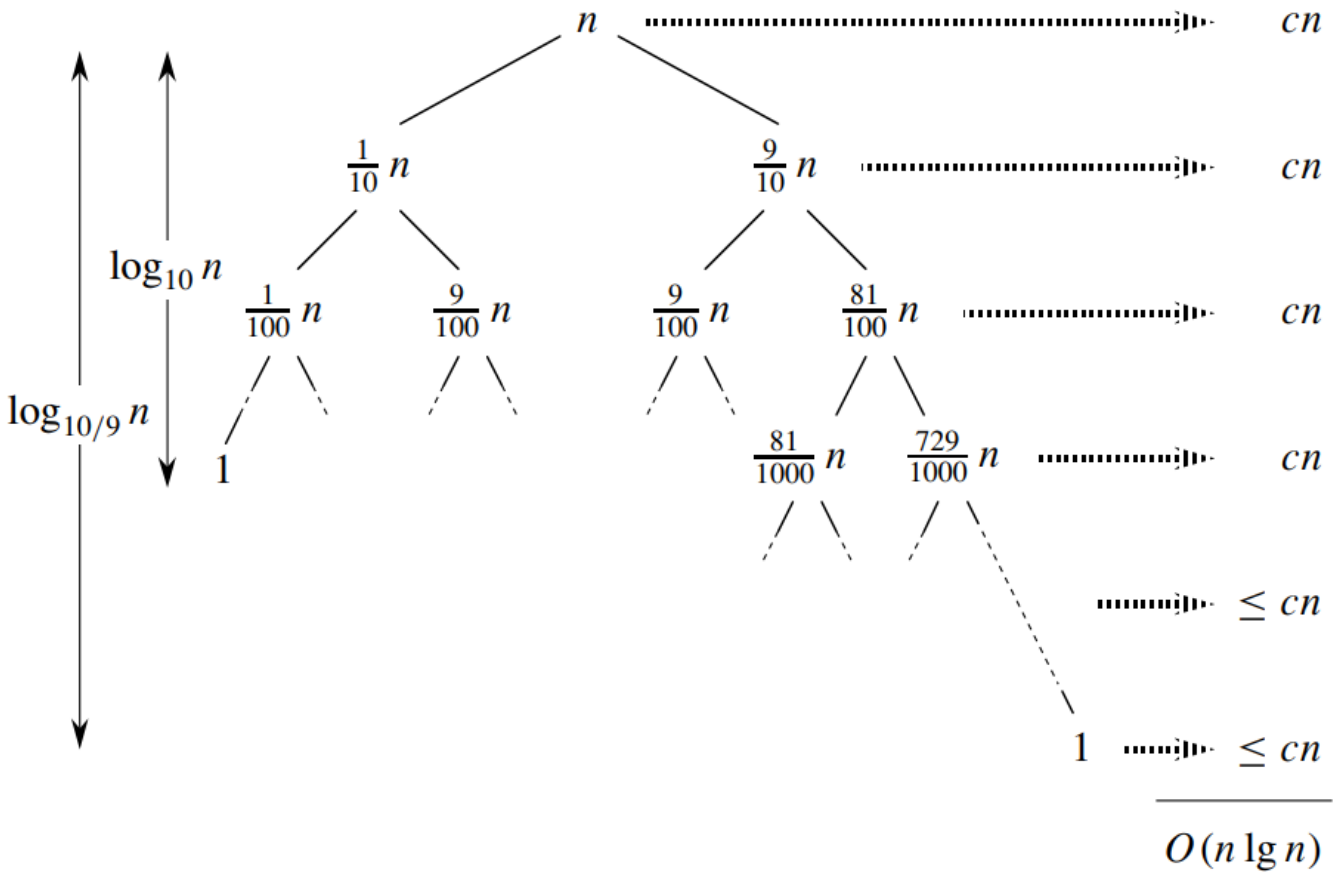


Figure 7.4 A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

Intuition behind average case

- Behavior of quicksort is determined by relative ordering of the values in the array elements given as input
- Using random array, there will be a mix of good and bad splits (some will be well-balanced, others will be unbalanced)
- On average, partition produces a split more balanced than 9:1 80% of the time, and a split less balanced than 9:1 20% of the time
- Suppose for the sake of example that we alternate between best-case splits and worst-case splits
- In (a), splits at two consecutive levels of recursion tree are shown. At root is cost n for partitioning, and subarrays produced have sizes $n-1$ and 0, which is the worst case
- At next level, subarray of size $n-1$ is best-case partitioned into subarrays of size $\frac{n-1}{2}-1$ and $(n-1)/2$
- Assuming cost of 1 for subarray size 0, this combination of bad split followed by good split produces 3 subarrays of sizes 0, $\frac{n-1}{2}-1$, and $(n-1)/2$, with a combined partitioning cost of $\Theta(n)+\Theta(n-1)=\Theta(n)$
- This is no worse than the situation in (b), where a single level of partitioning produces two subarrays of size $\frac{n-1}{2}$.
- intuitively, $\Theta(n-1)$ cost of bad split can be 'absorbed' into $\Theta(n)$ cost of good split, and resulting split is good.
- Thus, running time of quicksort when levels alternate between good and bad splits is still $O(nlogn)$, albeit with slightly higher constant

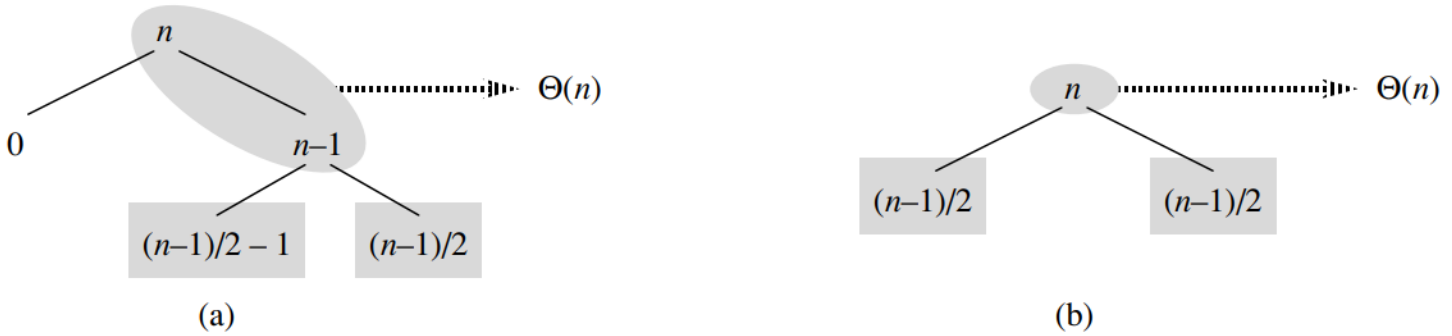


Figure 7.5 **(a)** Two levels of a recursion tree for quicksort. The partitioning at the root costs n and produces a "bad" split: two subarrays of sizes 0 and $n-1$. The partitioning of the subarray of size $n-1$ costs $n-1$ and produces a "good" split: subarrays of size $(n-1)/2-1$ and $(n-1)/2$. **(b)** A single level of a recursion tree that is very well balanced. In both parts, the partitioning cost for the subproblems shown with elliptical shading is $\Theta(n)$. Yet the subproblems remaining to be solved in (a), shown with square shading, are no larger than the corresponding subproblems remaining to be solved in (b).

Randomized quicksort

- In many cases, the input we want to sort is already sorted or in reverse-sorted order
- Randomized quicksort makes small modification: instead of using $A[r]$ as pivot, use a randomly chosen element from subarray $A[p..r]$
- Because this pivot element is randomly chosen, can expect split of input array to be reasonably well-balanced on average

```
RANDOMIZED-PARTITION(A, p, r)
1  i ← RANDOM(p, r)
2  exchange A[r] ↔ A[i]
3  return PARTITION(A, p, r)
```

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2     then q ← RANDOMIZED-PARTITION(A, p, r)
3          RANDOMIZED-QUICKSORT(A, p, q − 1)
4          RANDOMIZED-QUICKSORT(A, q + 1, r)
```